Modelling of Software intensive systems

Assignment 3 Petri nets analysis

First Edition

L. Jason	Jason.Liu@student.uantwerpen.be	s0213082
S. Kin Ning	KinNing.Shum@student.uantwerpen.be	s0202054

2024 - 12 - 21

Contents

1 | Solution

Here we will present our solution of the problem.

1.1 | Creation

Our schema consists of three main components: the waiting area, the berth, and the clock sequence.

The clock serves as the starting point, initializing itself along with the other constraints of the full port schema. Once the clock is initialized, the schema adjusts based on the input variables mm and nn (referred to as `workers` and `passageSize` in our case).

After initializing the clock, we can proceed with the simulation. The simulation begins in the waiting area, where ships are generated and then move through various stages to eventually reach the berth component.

In the berth component, we simulate its behavior as it serves ships. A ship can only enter the berth when it is free and can only be served if it is the ship's turn. This condition is met when the other berth is also active. Once the berth process is complete, the ship returns to the waiting area, where it can eventually reach the served state. These steps are clearly outlined by the clock's progression.

Now, let's delve into the steps involved in a single simulation. We use a backwards simulation approach, starting from the served state and working back to the generator state for each clock cycle. This ensures that each ship advances only one step at a time.

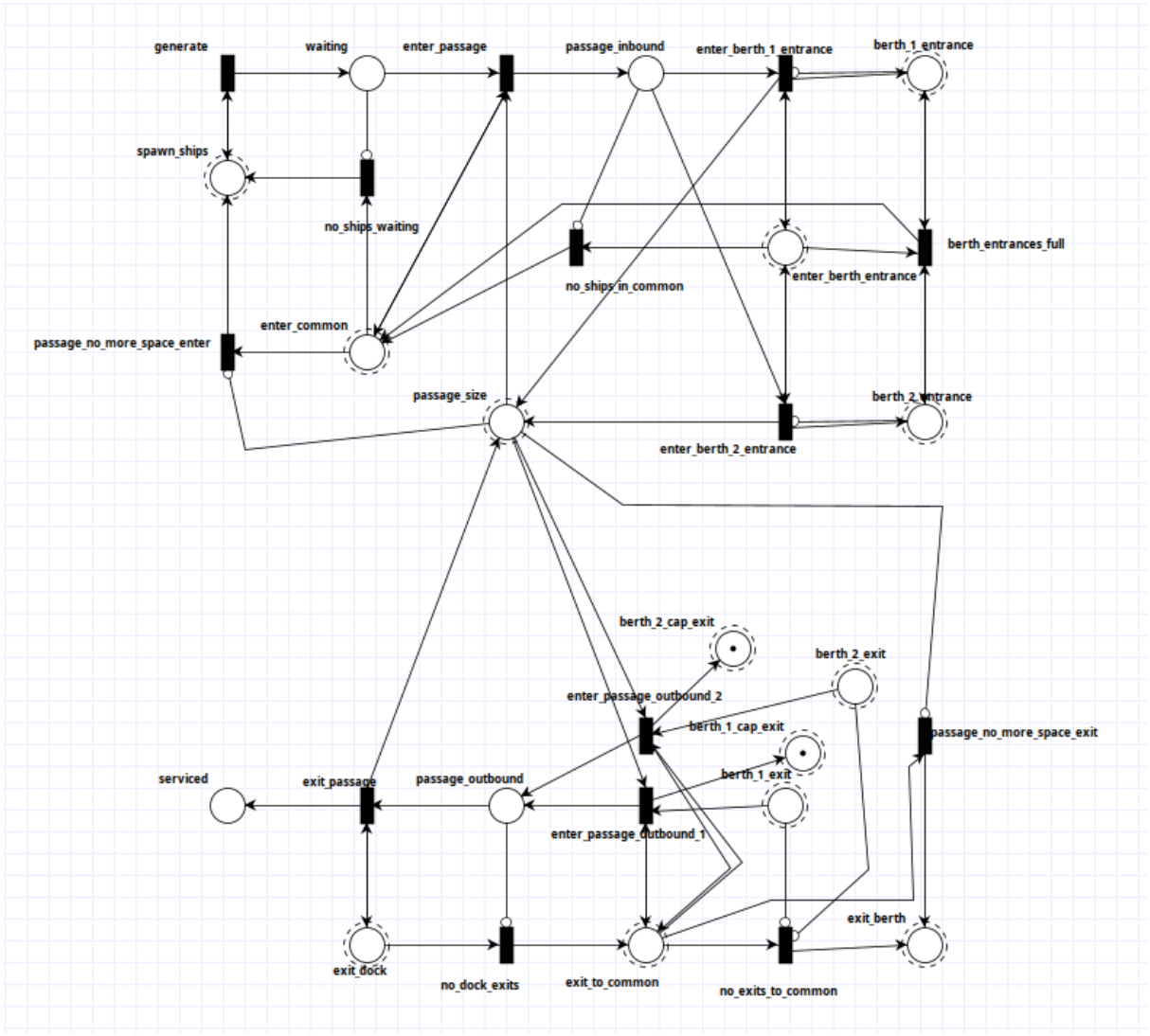

Figure 1: Waiting area

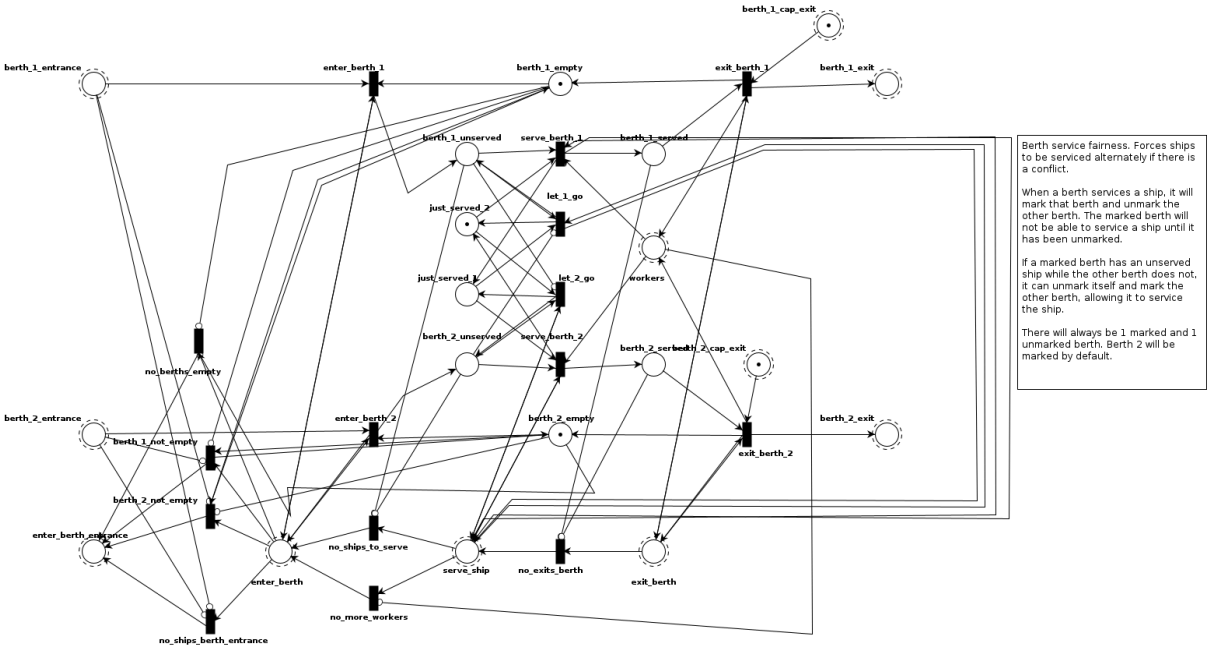

Figure 2: Berth

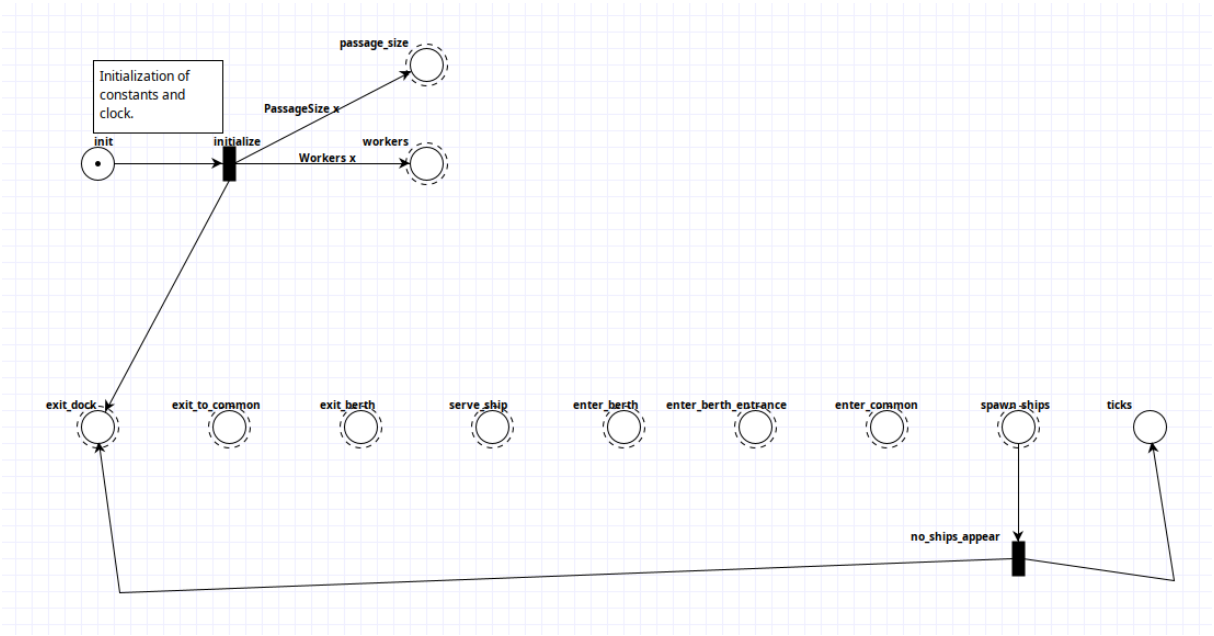

Figure 3: Clock

We use the clock primarily to track the different stages of the simulation. A deeper understanding, however, comes from examining the schema itself. The clock is integrated into the schema, and a good example of this can be seen in the berth component.

If we focus on the bottom half of the berth component, we can observe the various stages it goes through, moving from right to left. First, ships exit the berth if necessary. Next, ships waiting in the berth are served. Then, new ships enter the berth if space is available. Finally, the remaining stages handle the movement of ships through their respective steps.

The same behavior can be observed in the waiting area component.

In general, we use a token mechanism to control which transitions are allowed to fire, ensuring that the simulation progresses logically and accurately.

All the components can be found in the `DockingStation.tapn` file.

1.2 | Traces

We observe two different scenarios that can happen depending on the settings of the schema.

1.2.1 | 1 worker and common passage size of one

This can be found in the `DockingStation_1_1.trc` file. This trace will eventually lead to a deadlock.

Why is this? This is because our passage size has a capacity of 1, when the ships exit the common passage is already occupied making it impossible to leave.

1.2.2 | 4 workers and common passage size of three

This can be found in the `DockingStation_4_3.trc` file. This trace will just work normally. There is no race condition met during running of this trace.

This is because we have enough space in the common passage after leaving the berth. So, there is no instance where it could get stuck.

1.3 | Analysis

The analysis is done with m = 3 and n = 1.

1.3.1 | Reachability

First, let us discuss the reachability graph.

We run the following command:

```
python3 RC.py DockingStation.tapn DockingStationR.dot R -I 20
```

This gets us the following reachability graph:

Figure 4: Reachability graph

If we run this without any iterations, this would lead to a huge file with a repeating part, that you can clearly already see in this graph.

The graph is infinite because there is no constraint on how many ships can be generated. This leads to an infinite graph because there is no end. A way to fix this is to add a termination condition. Not only that we have an infinite loop of no ships that can appear, if we

make this deterministic and say that every tick one ship must appear, then this loop will become finite.

We will not generate the graph in `png` format, we did make a `dot` format of this reachability graph, called `DockingStationRFinite.dot`. The graph is based on `DockingStationFinite.tapn`.

The changes are based on the things that could lead to infinite behavior that we mentioned before. By removing some transitions that are not deterministic and adding a terminal condition, the infinite behavior is gone. The simulation is still long because it has to go numerous clock ticks to eventually end.

1.3.2 | Coverability

The coverability graph looks something like this:

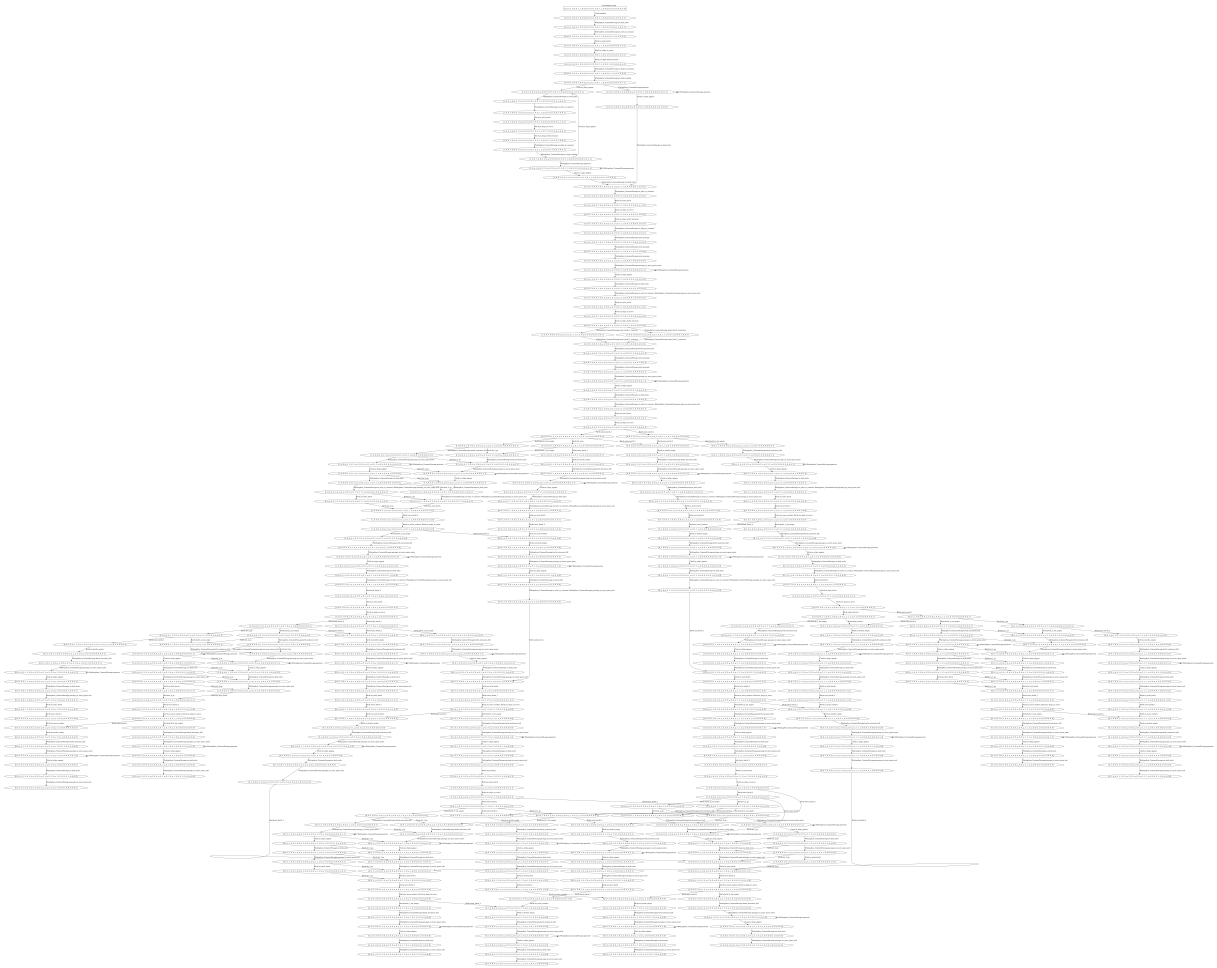

Figure 5: Coverability graph

Not readable in the `pdf` but if we analyze the patterns closely we can note the following:

1. Based on the graph, we can observe that our model is not bounded. This conclusion arises from the presence of ω, which indicates a state that can progress indefinitely.
2. Repetitive sequences are also evident in the graph. This is likely due to the clock's continuous looping, causing certain sequences to repeat.
3. Additionally, we can identify several instances of deadlock. This can be verified by checking for leaf states —states without any outgoing edges— which indicate potential deadlock situations.

4. We can also infer partial liveness in the system. The presence of loops in the graph suggests that the model could exhibit liveness under certain scenarios.

1.3.3 | Invariant Analysis

```
P-Invariants:
  M(Berth.berth_1_served) + M(Berth.berth_2_served) + M(Clock.init) + M(workers) = 1
  M(Clock.init) + M(enter_berth) + M(enter_berth_entrance) + M(enter_common) +
M(exit_berth) + M(exit_dock) + M(exit_to_common) + M(serve_ship) + M(spawn_ships) = 1
  M(Berth.berth_1_empty) + M(Berth.berth_1_served) + M(Berth.berth_1_unserved) = 1
  M(Berth.just_served_1) + M(Berth.just_served_2) = 1
  M(Berth.berth_2_empty) + M(Berth.berth_2_served) + M(Berth.berth_2_unserved) = 1
  3 * M(Clock.init) + M(WaitingArea_CommonPassage.passage_inbound) +
M(WaitingArea_CommonPassage.passage_outbound) + M(passage_size) = 3
  M(berth_2_cap_exit) + M(berth_2_exit) = 1
  M(berth_1_cap_exit) + M(berth_1_exit) = 1
```

Let us look at them step by step. The first one is obvious because we only have one worker, so only one ship can be served at a time. This can be seen in this line too, because all of it can only be one.

The second one just explains the life cycle of one ship and how it would iterate through time.

The next one states what our berth can be. We can clearly see that it can have three different states over time.

The next one is our fairness token, that states which one can be active at the given time. We only want one to have priority, so that is why we only have one token.

Again we have our states but now for berth 2.

Next, we have an interesting invariant. It is the invariant of the max capacity of the common passage. It is shared between three places, this is because we have a capacity place and two places that the common space share with.

At last, we have the capacity constraints. This could have been avoided by making use of inhibitor arcs.

1.3.3.1 | Changes to the model

We can do many changes to change the invariant. For instance, adding extra places to the life cycle. This increases the amount of places it needs to share it with. Let us say we need an extra intermediate state before we enter the berth, this adds another state that the token needs to be passed with.

Or deleting the capacity of the exit. This eliminates this invariant because now there is no limit to how many can enter the exit. This would also solve the problem of deadlock, if too many ships arrive for the speed the workers can handle the ships.

Let us just try deleting the exit capacities. We get the following P invariants, and we can clearly tell, some things have changed.

```
P-Invariants:
  M(Berth.berth_1_served) + M(Berth.berth_2_served) + M(Clock.init) + M(workers) = 1
  M(Clock.init) + M(enter_berth) + M(enter_berth_entrance) + M(enter_common) +
M(exit_berth) + M(exit_dock) + M(exit_to_common) + M(serve_ship) + M(spawn_ships) = 1
  M(Berth.berth_1_empty) + M(Berth.berth_1_served) + M(Berth.berth_1_unserved) = 1
```

```
  M(Berth.just_served_1) + M(Berth.just_served_2) = 1
  M(Berth.berth_2_empty) + M(Berth.berth_2_served) + M(Berth.berth_2_unserved) = 1
  M(WaitingArea_CommonPassage.passage_outbound) +
M(WaitingArea_CommonPassage.serviced) + M(berth_1_cap_exit) + M(berth_1_exit) +
M(berth_2_cap_exit) + M(berth_2_exit) = 2
  3 * M(Clock.init) + M(WaitingArea_CommonPassage.passage_inbound) +
M(WaitingArea_CommonPassage.passage_outbound) + M(passage_size) = 3
```

The exit capacity now got removed, and we can also see that in the invariants. The invariants now don't specify that at most 1 ship can be at that current place. Also, we can see that the path between the common passage and exit can now have a total of 2 ships, this is because the exit can have one and the entry can have one.

This model is generated based on file `DockingStationPInvariant.tapn`.

1.3.4 | Boundedness

We checked boundedness with 20 tokens and 100. Both resulted that it was unbounded. This was to be expected, we reach this conclusion when we were analyzing our reachability graph.

Again, we know that several parts of our model can reach to infinity, because there is so no constraint on how many tokens can be in the "waiting" area.

We can analyze the same thing when we make use of queries to check if certain parts are bounded. If we do this for the P-Invariants, we can see that these are always bounded.

We can do this by stating the following safety property: $\Box$ (property that needs to hold at any given time). We can fill the brackets with anything that we want to check.

Let us do this for one of the P-invariants: $\Box(\text{berth\_1\_cap\_exit} + \text{berth\_1\_exit} = 1)$. This query is satisfied at any given time, no matter how many tokens we feed in. We can do this for the other P-invariants too, and we would get similar results.

We check on different parts of the system. Let us check on waiting and serviced. These are the parts that do not have a capacity. We can do this by stating $\Box(\text{WaitingArea\_CommonPassage.waiting} < 20)$. Let us say we want at most 20 in this place. When we check this, it gives us that the query is unsatisfied. We expected this because we knew it would exceed this. Now let us do this for serviced. Again we make use of similar query but now instead we make use of the serviced place and max 10 instead of 20. This also gives us unsatisfied, as expected.

All queries for Boundedness can be found in `DockingStation.tapn` prefixed with `1_`.

1.3.5 | Deadlock

Using the query `EF deadlock`, we can find a deadlock in our net. This deadlock can be reached using the trace `assets/traces/DockingStation_deadlock.trc`. There were at least 7 extra tokens needed for the query to find this deadlock. Using different numbers of extra tokens results in different amounts of transitions.

Extra tokens	7	8	9	10	11	12+
Transitions	7707	3662	3999	2923	2548	2549

More tokens generally mean fewer transitions, and more than 12 extra tokens gives the same results.

The cause of the deadlock is the size limit of the passage. Ships are not able to enter the outbound passage when the passage is full, causing them to remain in the berth exit. If there is a ship trying to exit a berth, it is unable to, as the berth exit is already occupied. Our flawed net thinks a ship can exit the berth, so the clock will not change phases until the ship exits. However, the ship actually can't exit the berth, causing a deadlock. When making the net, we did not realize this situation could occur. For some reason, we thought a ship could always exit a berth and didn't make appropriate transitions to prevent this deadlock.

I have recreated the deadlock situation, and it only takes 92 transitions, instead of 2549, to reach it. The simulation trace can be found in `assets/traces/DockingStation_deadlock_smaller.trc`. Note that both deadlock traces are from the main TAPAAL file (`DockingStation.tapn`) and not the deadlock one.

An easy but impractical way is to remove the size limits of the passage or the berth exits. There are many reasons why this is a bad idea. Another way is to make the clock be able to change phases when the berth exits are occupied. This is implemented in the file `DockingStationDeadlock.tapn`. While this fixes the deadlock issue, another issue arises. The clock just keeps changing phases without any ships actually moving, an infinite loop.

The best solution, in my opinion, is to reserve 1 spot in the passage for berth exits. This solves the deadlock and the infinite loop, as both are caused by ships not being able to enter the shared passage to leave the docking station. If the passage size is only 1, the passage should be used to alternate between entering and exiting the berth.

As mentioned in the coverability section, each leaf node in the reachability and coverability graph is a deadlock, as our net has no final state. In the coverability graph, each leaf node is exactly the situation this deadlock describes. The leaf node is reached with the transition `WaitingArea_CommonPassage.passage_no_more_space_exit`, just like the deadlock described above (not explicitly stated).

As for the finite reachability graph, this deadlock is never reached. Using the query `EF WaitingArea_CommonPassage.passage_inbound = 3` in `DockingStationFinite.tapn`, we notice that this property is never satisfied. This means that the passage is never occupied by inbound ships, allowing ships to exit berths, meaning that the deadlock never occurs. A leaf node in the reachability graph represents a deadlocks. The 'Terminal state' we introduced in the net to make the reachability graph finite is a deadlock we created.

1.3.6 | Liveness

We analyze the liveness of the transition `serve_berth_1` and `generate`.

For the first transition, the queries used are:

- L0: `Not L1`
- L1: `EF Berth.serve_berth_1`
- L2: `EF ((EF Berth.serve_berth_1) and Berth.berth_1_served = 1 and (EF !(EF Berth.serve_berth_1)))`
- L3: `EG (EF Berth.serve_berth_1)`
- L4: `AG (EF Berth.serve_berth_1)`

A transition is L1 if it can be fired at least once. L0 and L1 are therefore trivial. The query for L1 holds true.

As for L2, my reasoning is that there should be a state where the transition would be enabled AND it has already fired (`Berth.berth_1_served != 0`) AND there is a state where this transition is not reachable. In other words, it should be able to fire, and after firing, should be able to fire again or not be reachable any more. Meaning it could be fired 1, 2, 3, … times. Some other sources suggest that `AF(Berth.serve_berth_1)` can be used to test L2-liveness. However, it does not make much sense.

For L3, EG means that there exists a path where φ holds true in every state along that path. So the query says that there is a path where the transition can always be fired, meaning it can fire infinitely. The query takes too long so I do not know if it returns true or not.

Lastly, for L4, this transition will be able to fire in every state possible. this query also takes too long.

I do not have enough experience to know if these queries are correct, that is why I will deduce the liveness logically.

L0 and L1 are trivial. L2 should be true, as you can use berth 1 k times and then use berth 2 from then on. As our model has a deadlock (or an infinite loop if the deadlock is fixed), reaching the deadlock after using k times, proves that this transition is L2. L3 also holds true, the deadlock can be avoided if we do not fill up the passage size. Since there is no final state, this transition can be fired infinitely. It is not L4, as the deadlock can be reached without ever using berth 1, by using berth 2 instead.

Now the generate transition. The queries can be found in the `DockingStationDeadlock.tapn` file. This transition makes ship appear, so L1, L2 and L3 holds true. It is L3-live, because you can fire this transition as many times as you want, without the clock changing phases. If L3 is true, L2 is also true. As for L4, without spawning ships, the deadlock can't be reached. However, you can actually go into an infinite loop, without ever spawning a ship. Meaning there is a path where generate is never fired, so it is false.

Using a single arc, a transition can be killed. Generate can be killed by drawing an arc from the waiting area to generate. Now it needs a ship to make a ship, so if there is no ship, no new ships appear. As for the other transition, it needs a token from one of the phases of the clock. If we draw an arc from a different phase of the clock to the transition, it will need the clock to be in 2 phases for it to fire. This will never happen, as the clock can only be in one phase.

1.3.7 | Fairness

There are two notions of fairness, Berths are served alternatively if there are limited workers, and ships can only move one step per clock tick.

First, the berths fairness. We have added 2 places and 2 transitions. The places keep track of which berth has last been serviced. The transitions allow a berth to be serviced even if the other berth gets priority only if there are no ships to be serviced in the other berth.

What happens is when a berth gets serviced, the newly added places keep track of it, and will allow the other berth to get priority in the next clock tick. If a berth with priority does not have any ships to be serviced, but the other berth does, then the serviceable berth can go anyway.

For the other notion, our clock can only be at one phase at a time. It initializes at `exit_dock` and the phases only transition in one direction. The clock has to get through all the other phases before it can transition back to `exit_dock`. Every time it reaches this transition, it counts as a new cycle, and the tick count is incremented. To ensure the ships can only move one step at a time, the clock phases are implemented backwards. For example, when a ship enters a berth, its next step is to get serviced. However, the berth service phase goes before the enter berth phase, meaning after entering a berth, the ship can only be serviced in the next clock cycle.

To summarize, a ship can only move one step per tick, because
- The clock can only be in one phase at a time. Ships can only do a step in the correct clock phase.
- The clock phases are implemented backwards. The next step of a ship is the previous phase of the clock. It has to wait a full clock cycle before being able to do the next step.

There is also the fact that the ship movement is deterministic. If a ship can do a step, it must do that step. The clock phase won't change until no more steps can be done.

1.3.8 | Safety

If there are more ships than the capacity allows, they crash. Our net ensures the capacity of a place is never exceeded. There are 7 places with a maximum capacity:
- Common Passage
- berth entrance 1 and 2
- berth 1 and 2
- berth exit 1 and 2

The common passage has a variable capacity, but in this case, we shall set it to 3. The other places can only hold 1 ship. The following queries have been used in the `DockingStation.tapn`:
- `AG WaitingArea_CommonPassage.passage_inbound + WaitingArea_CommonPassage.passage_outbound <= 3`
- `AG (berth_1_entrance <= 1 and berth_2_entrance <= 1)`
- `AG (berth_1_exit <= 1 and berth_2_exit <= 1)`
- `AG (Berth.berth_1_unserved + Berth.berth_1_served <= 1 and Berth.berth_2_unserved + Berth.berth_2_served <= 1)`

These queries are all satisfied in our net. Except for the berth entrances property, where it says it is satisfied, but it keeps asking for more tokens. We can also logically explain why the berth entrances satisfies the property. The only transitions that puts a token in a berth entrance, have an inhibitor arc pointing from the berth entrance towards the transition. This means that there can only be a token placed in a berth entrance if it is empty. That is why it would never have more than 1 token.

We can make 2 ships crash by modifying our net to have an inhibitor arc removed. It is the inhibitor arc pointing from `berth_1_entrance` to `enter_berth_1_entrance`. If we verify the berth entrance query, it will show that the property is not satisfied, meaning a crash happens. The trace can be found in `assets/traces/DockingStation_crash.trc` for the `DockingStationCrash.tapn` file.

Another way to crash is to change the initial marking. By adding a token to `berth_2_empty`, we make our net think the net has a capacity of 2. Testing the query for the berth capacity will

result in not satisfied. This change is also made in `DockingStationCrash.tapn`. The trace can be found in `assets/traces/DockingStation_crash.trc`.